Collection Framework

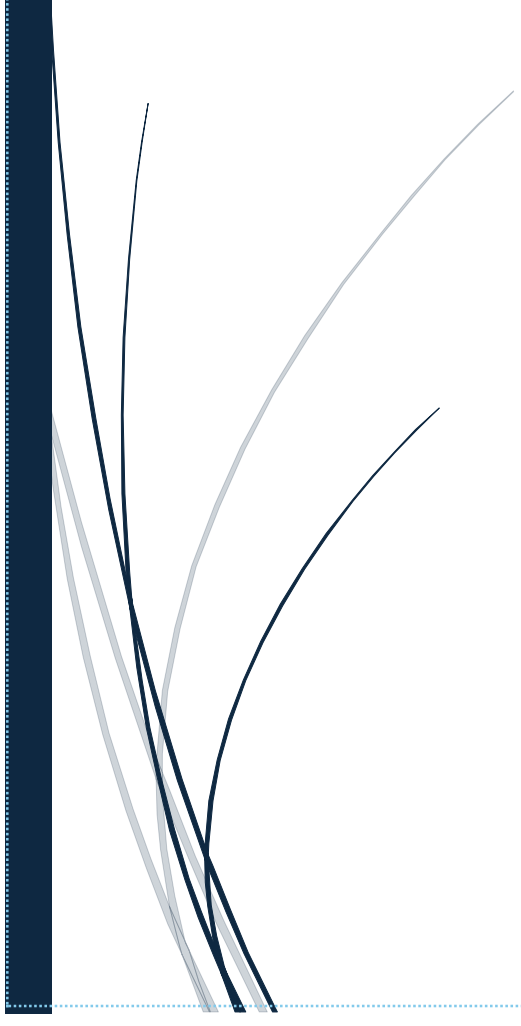
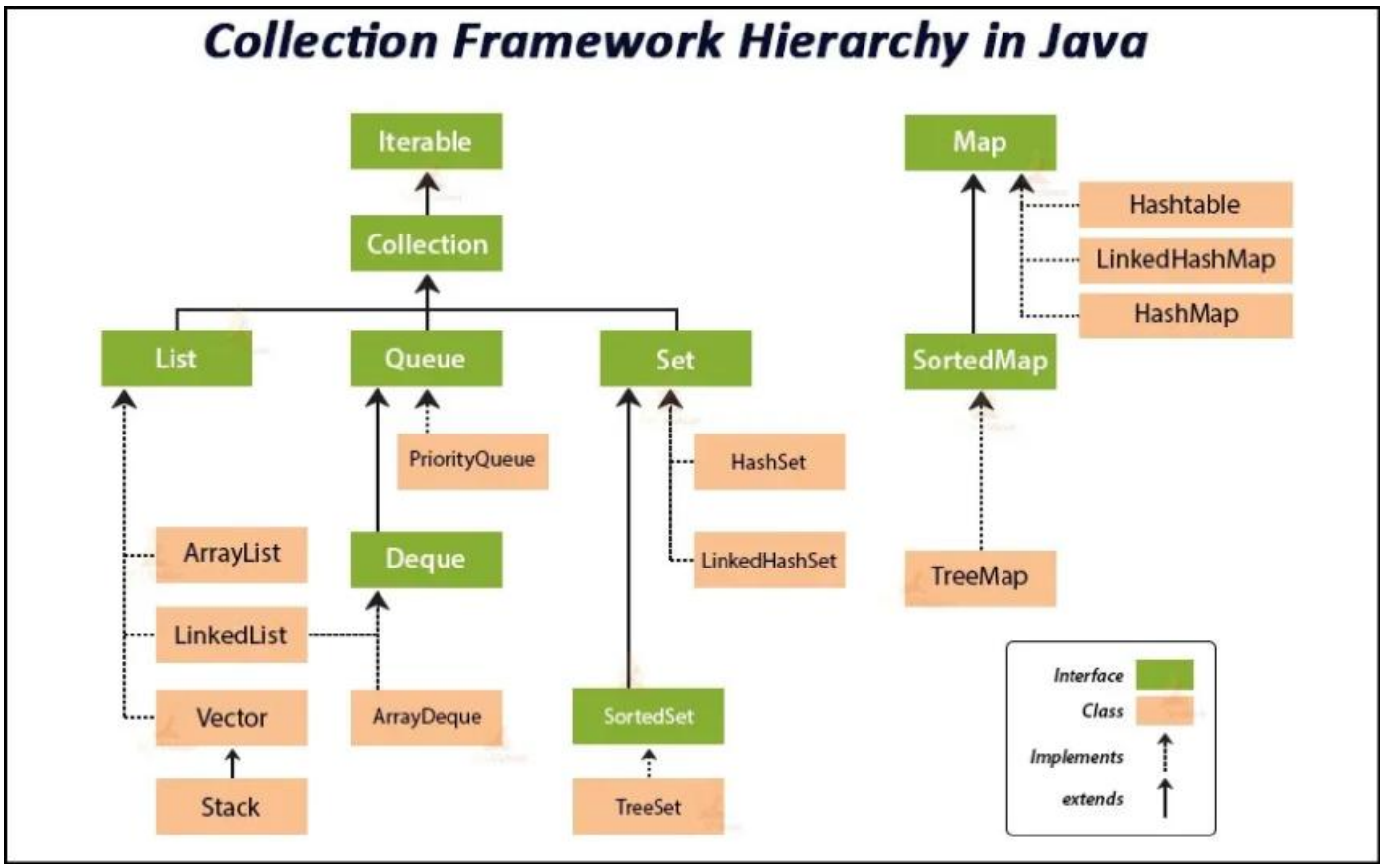


Table of Contents



An Array is an Indexed Collection of Fixed Number of Homogeneous Data Elements. The Main Advantage of Arrays is we can Represent Multiple Values by using Single Variable so that Readability of the Code will be Improved.

Limitations Of Object Type Arrays

- Arrays are Fixed in Size that is Once we created an Array there is No Chance of Increasing OR Decreasing Size based on Our Requirement. Hence to Use Arrays Concept Compulsory we should Know the Size in Advance which May Not be Possible Always.
- Arrays can Hold Only Homogeneous Data Type Elements.

```
Student[] s = new Student[10000];
s[0] = new Student(); // ✓ Allowed
s[1] = new Customer(); // ✗ Compile-time error
```

We can Resolve this Problem by using Object Type Arrays.

```
Object[] a = new Object[10000];
a[0] = new Student(); // ✓ Allowed
a[1] = new Customer(); // ✓ Allowed
```

- Arrays Concept is Not implemented based on Some Standard Data Structure Hence Readymade Methods Support is Not Available. Hence for Every Requirement we must write the Code Explicitly which Increases Complexity of the Programming. To Overcome Above Problems of Arrays we should go for Collections.

Advantages Of Collections

- Collections are Growable in Nature. That is based on Our Requirement we can Increase OR Decrease the Size.
- Collections can Hold Both Homogeneous and Heterogeneous Elements.

- Every Collection Class is implemented based on Some Standard Data Structure. Hence for Every Requirement Readymade Method Support is Available. Being a Programmer we must Use those Methods and we are Not Responsible to Provide Implementation.

Differences Between Arrays And Collections

Aspect	Arrays	Collections
Size	Fixed in Size	Growable in Nature
Memory Recommendation	Not Recommended to Use	Recommended to Use
Performance Recommendation	Recommended to Use	Not Recommended to Use
Data Elements	Only Homogeneous Data Elements	Both Homogeneous and Heterogeneous Elements
Underlying Data Structure	Not implemented based on Some Standard Data Structure	For every Collection class underlying Data Structure is Available
Readymade Method Support	Not Available	Available for Every Requirement
Types Held	Both Primitives and Objects	Only Objects but Not Primitives

1) Collection (I)

- It is the **root interface** of the Java Collections Framework.
- If we want to represent a **group of individual objects as a single entity**, then we should go for **Collection**.
- **Sub interfaces** include List, Set, and Queue.
- Does **NOT** include Map because Map represents key–value pairs, not individual elements.

Methods in Collection Interface

- The **Collection interface** defines the most common methods applicable to all collection objects.
- There is No Concrete Class which implements Collection Interface Directly.
- There is No Direct Method in Collection Interface to get Objects.

BELOW IS THE LIST OF METHODS:

1. **boolean add(Object o)** – Adds the specified element to the collection.
2. **boolean addAll(Collection c)** – Adds all elements from the specified collection.
3. **boolean remove(Object o)** – Removes the specified element from the collection.
4. **boolean removeAll(Collection c)** – Removes all elements present in the specified collection.
5. **boolean retainAll(Collection c)** – Removes all elements except those present in the specified collection.
6. **void clear()** – Removes all elements from the collection.
7. **boolean contains(Object o)** – Checks if the collection contains the specified element.
8. **boolean containsAll(Collection c)** – Checks if the collection contains all elements of the specified collection.
9. **boolean isEmpty()** – Checks if the collection is empty.
10. **int size()** – Returns the number of elements in the collection.
11. **Object[] toArray()** – Converts the collection into an array.
12. **Iterator iterator()** – Returns an iterator to traverse the collection.

2) List (I)

- It is the **child interface of Collection**.
- If we want to represent a **group of individual objects as a single entity** where **duplicates are allowed** and **insertion order is preserved**, then we should go for **List**.
- Provides **positional access** to elements using index.
- Common implementations: ArrayList, LinkedList, Vector, Stack.

Methods in List Interface

- The **List interface** defines the following specific methods for positional access and manipulation:

1. **void add(int index, Object o)** – Inserts the specified element at the given position.
2. **boolean addAll(int index, Collection c)** – Inserts all elements from the specified collection starting at the given position.
3. **Object get(int index)** – Returns the element at the specified position.
4. **Object remove(int index)** – Removes the element at the specified position.
5. **Object set(int index, Object new)** – Replaces the element at the specified position with the given element and returns the old element.
6. **int indexOf(Object o)** – Returns the index of the first occurrence of the specified element.
7. **int lastIndexOf(Object o)** – Returns the index of the last occurrence of the specified element.
8. **ListIterator listIterator()** – Returns a ListIterator for bidirectional traversal and modification.

2.1) ArrayList (C)

- It is a **class that implements the List interface**.
- **Resizable array** implementation; grows dynamically as elements are added. **Random access** is fast because it uses an array internally . **Null values allowed**.
- **Duplicates allowed** and **insertion order preserved**, but **Insertion and removal** in the middle are **slow** (requires shifting elements).
- **Not synchronized**; use Collections.synchronizedList() for thread safety.

Constructors

1.ArrayList<String> AL = new ArrayList<>();

- Creates an Empty ArrayList Object with Default Initial Capacity 10.
- If ArrayList Reaches its Max Capacity then a New ArrayList Object will be Created with
  - New Capacity = (Current Capacity * 3/2) + 1

2.ArrayList l = new ArrayList(int initialCapacity);

- Creates a new ArrayList with the specified initial capacity but Initial capacity determines the size of the internal array at creation.
- This constructor helps reduce resizing overhead when you know the approximate number of elements in advance

3.ArrayList l = new ArrayList(Collection c);

- Creates a new ArrayList that contains all elements of the specified collection c, in the order returned by the collection's iterator.
- This is useful for:
  - Copying elements from another collection into a new ArrayList.

- Converting any collection type (e.g., Set, Queue) into a List.

```java
Collection<String> names = new HashSet<>();
names.add("Alice");
names.add("Bob");

ArrayList<String> list = new ArrayList<>(names); // Copies elements from HashSet
```

How to get Synchronized Version of ArrayList Object

By Default ArrayList Object is Non- Synchronized. To get a synchronized version of an ArrayList object in Java, you can use the **Collections.synchronizedList()** method. This method wraps your existing ArrayList in a thread-safe wrapper.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SynchronizedArrayListExample {
    public static void main(String[] args) {
        // Create a normal ArrayList
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        // Get synchronized version of the ArrayList
        List<String> synchronizedList = Collections.synchronizedList(list);

        // Accessing synchronized list safely
        synchronized (synchronizedList) {
            for (String item : synchronizedList) {
                System.out.println(item);
            }
        }
    }
}
```

- **Collections.synchronizedList()** returns a **thread-safe** wrapper around the given list.
- Important: Iteration over the synchronized list must be done inside a synchronized block to avoid **ConcurrentModificationException**.
- This approach is useful when multiple threads need to access and modify the same list.
- ArrayList is the Best Choice if we want to Perform Retrieval Operation Frequently.
- But ArrayList is Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle. Because it required Several Shift Operations Internally.

2.2) LinkedList (C):

- It is a class that implements both List and Deque interfaces.
- Doubly-linked list implementation; each element has links to previous and next nodes.
- Duplicates, Null values allowed and insertion order preserved.
- Insertion and removal are fast (especially in the middle) because no shifting of elements is required.
- Random access is slow compared to ArrayList because traversal is needed.
- Can be used as List, Queue, or Deque.
- Not synchronized; use Collections.synchronizedList() for thread safety.

Constructors:

1.LinkedList<String> list = new LinkedList<>();

- Creates an empty LinkedList with no initial elements.
- Default capacity concept does NOT apply because LinkedList uses nodes, not arrays.
- After creation, you can add elements using add(), addFirst(), or addLast().

2.LinkedList l = new LinkedList(Collection c);

- Creates a LinkedList containing all elements of the specified collection c, in the order returned by its iterator.
- Useful for:
    - Copying elements from another collection into a LinkedList.
    - Converting any collection type (e.g., ArrayList, HashSet) into a LinkedList.

```
1  Collection<String> names = new ArrayList<>();
2  names.add("Alice");
3  names.add("Bob");
4
5  LinkedList<String> list = new LinkedList<>(names); // Copies elements from ArrayList
```

Methods

Usually we can Use LinkedList to Implement Stacks and Queues. To Provide Support for this Requirement LinkedList Class Defines the following 6 Specific Methods.

- **void addFirst(Object o):**Inserts the specified element at the beginning of the list.
- **void addLast(Object o):**Appends the specified element at the end of the list.
- **Object getFirst():**Returns the first element in the list and throws NoSuchElementException if the list is empty.
- **Object getLast():** Returns the last element in the list and throws NoSuchElementException if the list is empty.
- **Object removeFirst():**Removes and returns the first element in the list and throws NoSuchElementException if the list is empty.
- **Object removeLast():**Removes and returns the last element in the list and Throws NoSuchElementException if the list is empty.

2.3) Vector(C)

- The Underlying Data Structure is Resizable Array OR Growable Array.
- Insertion Order is Preserved.
- Heterogeneous and Duplicate Objects are allowed.
- null Insertion is Possible.
- Implements Serializable, Cloneable and RandomAccess interfaces.
- Every Method Present Inside Vector is Synchronized and Hence Vector Object is Thread Safe.
- Vector is the Best Choice if Our Frequent Operation is Retrieval.
- Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.

Constructors

1. Vector<String> v = new Vector<>();

- Creates an empty Vector with a default capacity of 10 elements.
- Ideal when the number of elements is unknown, since the Vector can grow dynamically.

2. Vector<String> v = new Vector<>(20);

- Instantiates an empty Vector with a user-defined initial capacity.
- Useful for reducing resizing operations when the expected size is known.

3. Vector<String> v = new Vector<>(10, 5);

- Creates a Vector with a set initial capacity and a specified capacity increment (growth factor).
- When the Vector exceeds its capacity(10), it expands by the given increment(5), allowing control over memory management.

4. Vector(Collection c)

- Builds a Vector containing all elements from the provided collection.
- Handy for copying contents from other collections into a Vector during construction.

```
1  Collection<String> names = List.of("Alice", "Bob");
2  Vector<String> v = new Vector<>(names);
```

Methods

1) To Add Elements

- add(Object o) → Collection
- add(int index, Object o) → List
- addElement(Object o) → Vector

2) To Remove Elements

- remove(Object o) → Collection
- removeElement(Object o) → Vector
- remove(int index) → List
- removeElementAt(int index) → Vector
- clear() → Collection
- removeAllElements() → Vector

3) To Retrieve Elements

- Object get(int index) → List
- Object elementAt(int index) → Vector
- Object firstElement() → Vector
- Object lastElement() → Vector

4) Some Other Methods

- int size() → Returns number of elements.
- int capacity() → Returns current capacity of Vector.
- Enumeration elements() → Returns an Enumeration for traversing elements.

2.3.1) Stack (C)

- It is a **legacy class** that extends **Vector** and represents a **LIFO (Last-In-First-Out) stack**.
- Introduced in **Java 1.0**; later retrofitted into Collections Framework.
- **Duplicates allowed** and **insertion order preserved**.
- **Thread-safe** because it inherits synchronized methods from Vector.
- Commonly used for **stack operations** like push and pop.

Constructors

Stack<Integer> stack = new Stack<>();

- Creates an empty Stack.

Methods

1) push(E item): Adds an item to the **top of the stack**.

2) pop(): Removes and returns the **top element** of the stack and Throws EmptyStackException if the stack is empty.

3) peek():Returns the **top element** without removing it.

4) empty():Checks if the stack is empty.

5) search(Object o):Returns the **1-based position** of the element from the top of the stack, or -1 if not found.

```
1  import java.util.Stack;
2
3  class StackDemo {
4      public static void main(String[] args) {
5          Stack s = new Stack();
6          s.push("A");
7          s.push("B");
8          s.push("C");
9
10         System.out.println(s);              // [A, B, C]
11         System.out.println(s.search("A"));  // 3
12         System.out.println(s.search("Z"));  // -1
13     }
14 }
```

Cursors

- If we want to get Objects One by One from the Collection then we should go for Cursors.
- There are 3 Types of Cursors Available in Java.
  1. Enumeration
  2. Iterator
  3. ListIterator

1)Enumeration

- We can Use Enumeration to get Objects One by One from the Collection.
- We can Create Enumeration Object by using elements().
  - public Enumeration elements();

Eg: Enumeration e = v.elements(); //v is Vector Object.

Methods:

1. public booleanhasMoreElements();
2. public Object nextElement();

Limitations

- Enumeration Concept is Applicable Only for Legacy Classes and it is Not a Universal Cursor.
- By using Enumeration we can Perform Read Operation and we can't Perform Remove Operation.

```java
import java.util.*;

class EnumerationDemo {
    public static void main(String[] args) {
        Vector v = new Vector();
        for (int i = 0; i <= 10; i++) {
            v.addElement(i);
        }

        System.out.println(v); // Prints all elements in the Vector

        Enumeration e = v.elements();
        while (e.hasMoreElements()) {
            Integer l = (Integer) e.nextElement();
            if (l % 2 == 0)
                System.out.println(l); // Prints even numbers
        }

        System.out.println(v); // Prints the original Vector again
    }
}

```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
0
2
4
6
8
10
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

2)Iterator:

- We can Use Iterator to get Objects One by One from Collection.
- We can Apply Iterator Concept for any Collection Object. Hence it is Universal Cursor.
- By using Iterator we can be Able to Perform Both Read and Remove Operations.
- We can Create Iterator Object by using iterator() of Collection Interface.

  public Iterator iterator();

Eg: Iterator itr = c.iterator(); //c Means any Collection Object.

Methods:

- **boolean hasNext()** → Checks if more elements exist.
- **Object next()** → Returns the next element.
- **void remove()** → Removes the last returned element.

Limitations:

- By using Enumeration and Iterator we can Move Only towards Forward Direction and we can't Move Backward Direction. That is these are Single Direction Cursors but Not Bi-Direction.
- By using Iterator we can Perform Only Read and Remove Operations and we can't Perform Addition of New Objects and Replacing Existing Objects.

```java
import java.util.*;

class IteratorDemo {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        for (int i = 0; i <= 10; i++) {
            list.add(i);
        }

        System.out.println(list); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

        Iterator<Integer> itr = list.iterator();
        while (itr.hasNext()) {
            Integer n = itr.next();
            if (n % 2 == 0) {
                System.out.println(n); // Prints even numbers
            } else {
                itr.remove(); // Removes odd numbers
            }
        }

        System.out.println(list); // [0, 2, 4, 6, 8, 10]
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
0
2
4
6
8
10
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

3)ListIterator:

- ListIterator is the Child Interface of Iterator.
- By using ListIterator we can Move Either to the Forward Direction OR to the Backward Direction. That is it is a Bi-Directional Cursor.
- By using ListIterator we can be Able to Perform Addition of New Objects and Replacing existing Objects. In Addition to Read and Remove Operations.
- We can Create ListIterator Object by using listIterator().
    - public ListIterator listIterator();

Eg:

ListIterator ltr = l.listIterator(); //l is Any List Object

Methods:

ListIterator is the Child Interface of Iterator and Hence All Iterator Methods by Default Available to the ListIterator.

Forward Direction

- public boolean hasNext() → Checks if there is a next element.
- public Object next() → Returns the next element.
- public int nextIndex() → Returns the index of the next element.

Backward Direction

- public boolean hasPrevious() → Checks if there is a previous element.
- public Object previous() → Returns the previous element.
- public int previousIndex() → Returns the index of the previous element.

Modification Methods

- public void remove() → Removes the last returned element.
- public void set(Object new) → Replaces the last returned element.
- public void add(Object new) → Inserts a new element at the current position.

```java
import java.util.*;

class ListIteratorDemo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        System.out.println("Original List: " + list);

        ListIterator<String> itr = list.listIterator();

        System.out.println("Forward Traversal:");
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }

        System.out.println("Backward Traversal:");
        while (itr.hasPrevious()) {
            System.out.println(itr.previous());
        }
    }
}
```

Output:

```
Original List: [A, B, C]
Forward Traversal:
A
B
C
Backward Traversal:
C
B
A
```

THE MOST POWERFUL CURSOR IS LISTITERATOR. BUT ITS LIMITATION IS, IT IS APPLICABLE ONLY FOR LIST OBJECTS.

Property	Enumeration	Iterator	ListIterator
Applicable For	Only Legacy Classes	Any Collection Objects	Only List Objects
Movement	Single Direction (Only Forward)	Single Direction (Only Forward)	Bi-Direction
How To Get	By using elements()	By using iterator()	By using listIterator() of List (I)
Accessibility	Only Read	Read and Remove	Read, Remove, Replace And Addition of New Objects
Methods	hasMoreElements(), nextElement()	hasNext(), next(), remove()	9 Methods
Is it legacy?	Yes (1.0 Version)	No (1.2 Version)	No (1.2 Version)

3) Set

- It is the Child Interface of Collection.
- If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed and Insertion Order is Not Preserved then we should go for Set.
- Set Interface doesn't contain any New Methods and Hence we must Use Only Collection Interface Methods

3.1) HashSet (C)

- It is a class that implements the Set interface.
- Stores elements in a hash table (backed by HashMap).
- Duplicates NOT allowed and Insertion order NOT preserved (elements are unordered).
- Null values allowed (only one null element).
- Performance:
    - Operations like add(), remove(), contains() are O(1) on average.
- Not synchronized; use Collections.synchronizedSet() for thread safety.

Constructors

HashSet()

- HashSet<String> set = new HashSet<>();
- Creates an empty HashSet with default initial capacity (16) and load factor (0.75).

HashSet(int initialCapacity)

- HashSet<Integer> set = new HashSet<>(50);
- Creates an empty HashSet with specified initial capacity.

HashSet(int initialCapacity, float loadFactor)

- HashSet<String> set = new HashSet<>(20, 0.80f)
- Creates an empty HashSet with specified initial capacity and load factor.

HashSet(Collection<? extends E> c)

- List<String> list = Arrays.asList("A", "B", "C");
- Creates a HashSet containing all elements of the specified collection.

Load Factor:

Definition:

Load Factor is a measure of how full the hash table can get before it needs to resize.

It is the ratio:

$$\text{Load Factor} = \frac{\text{Number of elements stored}}{\text{Current capacity of the hash table}}$$

Default Value:

0.75 (75%) in Java for HashSet and HashMap.

Purpose:

- Controls trade-off between time and space.
- Lower load factor → More space, fewer collisions → Faster lookups.
- Higher load factor → Less space, more collisions → Slower lookups.

Resizing Behaviour:

- When the number of elements exceeds capacity × loadFactor, the hash table rehashes:
- Capacity is doubled. All elements are rehashed into the new table.

3.1.1)LinkedHashSet (C):

- Underlying Data Structure: Combination of LinkedList + Hashtable.
- Duplicates are Not allowed and Insertion Order is Preserved (maintains the order in which elements were inserted).
- Null Values: Allowed (only one null element).
- Performance: Slightly slower than HashSet due to maintaining insertion order.
- Not synchronized; use Collections.synchronizedSet() for thread safety.

Constructors

1)LinkedHashSet()

- `LinkedHashSet<String> lhs = new LinkedHashSet<>();`
- `Creates an empty LinkedHashSet with default capacity (16) and load factor (0.75).`

2)LinkedHashSet(int initialCapacity)

- `LinkedHashSet<Integer> lhs = new LinkedHashSet<>(50);`
- `Creates an empty LinkedHashSet with specified initial capacity.`

3)LinkedHashSet(int initialCapacity, float loadFactor)

- `LinkedHashSet<String> lhs = new LinkedHashSet<>(20, 0.80f);`
- `Creates an empty LinkedHashSet with specified initial capacity and load factor.`

4)LinkedHashSet(Collection<? extends E> c)

- `List<String> list = Arrays.asList("A", "B", "C");`
- `Creates a LinkedHashSet containing all elements of the specified collection.`

In General we can Use LinkedHashSet and LinkedHashMap to Develop Cache Based Applications where Duplicates are Not Allowed and Insertion Order Must be Preserved.

3.2) SortedSet:

- ❖ It is the Child Interface of Set.
- ❖ If we want to Represent a Group of Individual Objects without Duplicates and allObjects will be Inserted According to Some Sorting Order, then we should go for SortedSet.
- ❖ The Sorting can be Either Default Natural Sorting OR Customized Sorting Order.
- ❖ For String Objects Default Natural Sorting is Alphabetical Order.
- ❖ For Numbers Default Natural Sorting is Ascending Order.

Methods

1) Comparator<? super E> comparator()

- ❖ Returns the Comparator used to order the elements in the set.
- ❖ If the set uses natural ordering, this method returns null

2) E first()

- ❖ Returns the lowest (first) element in the set.
- ❖ Throws NoSuchElementException if the set is empty.

3) E last()

- ❖ Returns the highest (last) element in the set.
- ❖ Throws NoSuchElementException if the set is empty.

4) SortedSet<E> headSet(E toElement)

- ❖ Returns a view of elements less than toElement.

5) SortedSet<E> tailSet(E fromElement)

- ❖ Returns a view of elements greater than or equal to fromElement.

6) SortedSet<E> subSet(E fromElement, E toElement)

- ❖ Returns a view of elements between fromElement (inclusive) and toElement (exclusive).

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetNaturalOrderDemo {
    public static void main(String[] args) {
        // Natural ordering: Integer implements Comparable<Integer>
        SortedSet<Integer> set = new TreeSet<>();
        set.add(50);
        set.add(10);
        set.add(30);
        set.add(20);
        set.add(40);

        System.out.println("SortedSet (natural order): " + set); // [10, 20, 30, 40, 50]
        // 1) comparator()
        System.out.println("comparator(): " + set.comparator()); // null for natural order
        // 2) first()
        System.out.println("first(): " + set.first()); // 10
        // 3) last()
        System.out.println("last(): " + set.last()); // 50
        // 4) headSet(toElement): elements strictly less than 30
        SortedSet<Integer> head = set.headSet(30);
        System.out.println("headSet(30): " + head); // [10, 20]
        // 5) tailSet(fromElement): elements >= 30
        SortedSet<Integer> tail = set.tailSet(30);
        System.out.println("tailSet(30): " + tail); // [30, 40, 50]
        // 6) subSet(fromElement, toElement): [20, 50) -> 20 inclusive, 50 exclusive
        SortedSet<Integer> sub = set.subSet(20, 50);
        System.out.println("subSet(20, 50): " + sub); // [20, 30, 40]

    }
}
```

3.2.1) TreeSet:

- The Underlying Data Structure is Balanced Tree.
- Insertion Order is Not Preserved and it is Based on Some Sorting Order.
- Heterogeneous Objects are Not Allowed. If we are trying to Insert we will get Runtime Exception Saying ClassCastException.
- Duplicate Objects are Not allowed and null Insertion is Possible (Only Once).
- Implements Serializable and Cloneable Interfaces but Not RandomAccess Interface.

Constructors

TreeSet()

- TreeSet<Integer> ts = new TreeSet<>();
- Creates an empty TreeSet that uses natural ordering of elements.

TreeSet(Collection<? extends E> c)

- List<String> list = Arrays.asList("B", "A", "C");
- Creates a TreeSet containing all elements of the specified collection, sorted in natural order.

TreeSet(SortedSet<E> s)

- SortedSet<Integer> sortedSet = new TreeSet<>(Arrays.asList(1, 2, 3));
  TreeSet<Integer> ts = new TreeSet<>(sortedSet);
- Creates a TreeSet with the same elements and ordering as the specified SortedSet

TreeSet(Comparator<? super E> comparator)

- TreeSet<String> ts = new TreeSet<>(String.CASE_INSENSITIVE_ORDER)
- Creates an empty TreeSet that uses the specified Comparator for ordering.

```java
import java.util.TreeSet;

class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet();
        t.add("A");
        t.add("a");
        t.add("B");
        t.add("Z");
        t.add("L");

        t.add(new Integer(10)); // Causes ClassCastException
        t.add(null);            // Causes NullPointerException

        System.out.println(t);  // Expected output: [A, B, L, Z, a]
    }
}
```

null Acceptance

- For Empty TreeSet as the 1st Element null Insertion is Possible. But after inserting that null if we are trying to Insert any Element we will get NullPointerException.
- For Non- Empty TreeSet if we are trying to Insert null we will get NullPointerException.
- If we are Depending on Default Natural Sorting Order Compulsory Objects should be
- Homogeneous and Comparable. Otherwise we will get RE: ClassCastException.
- An object is said to be Comparable if and only if corresponding class implements Comparable interface.
- All Wrapper Classes, String Class Already Implements Comparable Interface. But StringBuffer Class doesn't Implement Comparable Interface.

3.2.2)NavigableSet(I)

- The java.util.NavigableSet interface extends the SortedSet interface to provide a sorted set with enhanced navigation capabilities.
- This interface is designed to maintain its elements in a sorted order, which can be determined by the elements' natural ordering or by a custom Comparator provided at set creation.
- Sorted Order: Elements are always maintained in a sorted sequence, either according to their natural order or as specified by a custom comparator.
- When natural ordering is used, NavigableSet does not permit null elements.

Methods

floor(e)

- Purpose: Returns the greatest element ≤ e.

lower(e)

- Purpose: Returns the greatest element < e.

ceiling(e)

- Purpose: Returns the least element ≥ e.

higher(e)

- Purpose: Returns the least element > e.

pollFirst()

- Purpose: Removes and returns the first (lowest) element.

pollLast()

- Purpose: Removes and returns the last (highest) element.

descendingSet()

- Purpose: Returns a reverse-order view of the set.

Comparable (I):

- Comparable Interface Present in java.lang Package and it contains Only One Method **compareTo().**
  - ***public int compareTo(Object o);***
- Compares the current object (obj1) with another object (obj2).
- Returns an integer indicating their relative order.

Negative (-ve):

- obj1 should come before obj2 in the sorted order.
- Example: "A".compareTo("B") → -1 (A comes before B).

Positive (+ve):

- obj1 should come after obj2.
- Example: "B".compareTo("A") → 1 (B comes after A).

Zero (0):

- obj1 and obj2 are equal in terms of ordering.
- Example: "A".compareTo("A") → 0.

NOTES

- Whenever we are Depending on Default Natural Sorting Order and if we are trying to Insert Elements then Internally JVM will Call **compareTo()** to Identify Sorting Order.
- If we are Not satisfied with Default Natural Sorting Order OR if Default Natural Sorting Order is Not Already Available then we can Define Our Own Sorting by using **Comparator** Object.

Comparator(I)

- it is present in java.util and it defines custom ordering for objects It contains 2 Methods **compare()** and **equals().**

public int compare(Object obj1, Object obj2);

- This is the core method of the Comparator interface.
- It compares two objects and determines their ordering.
- Return Values:
  - Negative (-ve):
    - obj1 should appear before obj2 in the sorted order.
  - Positive (+ve):
    - obj1 should appear after obj2.
  - Zero (0):
    - obj1 and obj2 are considered equal in terms of ordering.

public boolean equals(Object o);

- This method checks whether the current comparator is equal to another object.
- It is part of the Comparator interface but implementation is optional.
- When implementing Comparator, you must implement **compare()** because it defines the sorting logic.
- Implementing equals() is **optional** because:
  - Every class inherits equals() from Object by default.
  - You only override it if you need custom equality logic for your comparator.

Comparable vs Comparator

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Method	`compareTo(Object o)`	`compare(Object o1, Object o2)`
Sorting	Natural order	Custom order
Location	Inside the class	Outside the class
Multiple Sort	Not possible	Possible
Java 8	No major changes	Added `comparing()`, `thenComparing()`
Use Case	Single default sort	Flexible, multiple sort strategies

Set Implementations in Java

Feature	HashSet	LinkedHashSet	TreeSet
Package	java.util	java.util	java.util
Underlying DS	Hash Table	Hash Table + Linked List	Red-Black Tree (Self-balancing)
Order	No order (unordered)	Maintains **insertion order**	Maintains **sorted order**
Null Allowed?	Yes (one null element)	Yes (one null element)	No (throws NullPointerException)
Duplicates	Not allowed	Not allowed	Not allowed
Performance	O(1) for add/search	O(1) for add/search	O(log n) for add/search
Sorting	Not supported	Not supported	Natural order or custom Comparator
Best Use Case	Fast lookups	Predictable iteration order	Sorted data without duplicates

Map

- Map is Not Child Interface of Collection.
- If we want to Represent a Group of Objects as Key- Value Pairs then we should go for Map.
- Both Keys and Values are Objects Only.
- Duplicate Keys are Not allowed. But Values can be Duplicated.
- Each Key- Value Pair is Called an Entry.

Methods

V put(K key, V value)

- Adds/updates a key–value pair.
- If the key doesn't exist, inserts and returns null.
- If the key exists, replaces old value and returns the old value.

void putAll(Map<? extends K, ? extends V> m)

- Copies all entries from another map.
- Overwrites keys that already exist.

V get(Object key)

- Retrieves the value for a key.
- Returns the value if the key exists; otherwise null.
- Note: with maps that allow null values, null can mean either “no mapping” or “mapped to null”—use containsKey() to disambiguate.

V remove(Object key)

- Removes the mapping for a key.
- Returns the previous value, or null if the key wasn’t present.

boolean containsKey(Object key)

- Checks if the key exists.
- boolean containsValue(Object value)
- Checks if any entry has the given value.

boolean isEmpty()

- Returns true if the map has no entries.

int size()

- Number of key–value pairs.

void clear()

- Removes all entries.

Set<K> keySet()

- View of all keys (backed by the map).
- Removing from this set removes from the map.
- Iteration order depends on implementation (e.g., insertion order for LinkedHashMap, sorted for TreeMap).

Collection<V> values()

- View of all values (backed by the map).
- Removing a value via the iterator removes its entry from the map.
- Values can have duplicates.

Set<Map.Entry<K,V>> entrySet()

- View of all entries (key–value pairs), backed by the map.
- You can iterate and update values in place via entry.setValue().
- Removing an entry from the set removes it from the map.

Entry (I)

- Each Key- Value Pair is Called One Entry.
- Without existing Map Object there is No Chance of existing Entry Object.
- Hence Interface Entry is Define Inside Map Interface.

HashMap

- The Underlying Data Structure is Hashtable.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Heterogeneous Objects are allowed for Both Keys and Values.
- Insertion Order is not preserved and it is based on hash code of the keys.
- null Insertion is allowed for Key (Only Once) and allowed for Values (Any Number of Times).

Comparison

Feature	HashMap	Hashtable
Synchronization	No method in `HashMap` is synchronized.	Every method in `Hashtable` is synchronized.
Thread Safety	Multiple threads can operate on a `HashMap` simultaneously → **Not thread-safe**.	Only one thread can operate at a time → **Thread-safe**.
Performance	High performance (no synchronization overhead).	Lower performance (due to synchronization).
Null Handling	Allows **one null key** and multiple null values.	Does **not allow null keys or values** (throws `NullPointerException`).
Version	Introduced in **Java 1.2**, part of Collections Framework, **non-legacy**.	Introduced in **Java 1.0**, considered **legacy**.

BY DEFAULT HASHMAP IS NON- SYNCHRONIZED. BUT WE CAN GET SYNCHRONIZED VERSION OF HASHMAP BY USING SYNCHRONIZEDMAP() OF COLLECTIONS CLASS.

Constructors

HashMap()

- Map<String, Integer> scores = new HashMap<>();
- **What it does:** Creates an empty map with default initial capacity (16) and default load factor (0.75).
- **When to use:** General purpose; you don't expect huge data or special tuning needs.

HashMap(int initialCapacity)

- Map<String, Integer> scores = new HashMap<>();
- **What it does:** Creates an empty map with a given initial capacity and default load factor (0.75).
- The actual internal table size is the smallest power of two ≥ initialCapacity.
- Helps avoid rehashing if you have a good estimate of entries.
- **When to use:** You know approximately how many entries you'll store (e.g., reading 10k records).

HashMap(int initialCapacity, float loadFactor)

- Map<Long, String> bigMap = new HashMap<>(1_000_000, 0.9f);
- **What it does:** Creates an empty map with your chosen initial capacity and load factor.
- Load factor meaning: Triggers a resize when size > capacity * loadFactor.
    - Default 0.75 is a balance between space and speed.
    - Higher load factor (e.g., 0.9) ⇒ fewer resizes but longer buckets ⇒ potentially slower lookups.
    - Lower load factor (e.g., 0.5) ⇒ more space, shorter buckets ⇒ potentially faster lookups.
- **When to use:** You need to optimize for memory vs speed for a known workload.

HashMap(Map<? extends K, ? extends V> m)

- Map<String, Integer> mutableCopy = new HashMap<>(original);
- **What it does:** Creates a map and copies all entries from the provided map.
- Capacity behavior: Capacity sized to hold the entries without immediate resize (implementation typically rounds to a power of two ≥ entries / loadFactor).
- **When to use:** Quick cloning/migration between map types.

LinkedHashMap

- It is the Child Class of HashMap.
- Hash-based Map that also maintains a doubly-linked list across entries.
- Allows one null key and multiple null values (like HashMap).
- O(1) average for put, get, remove (with small overhead for link maintenance).
- That is Insertion Order is Preserved.

In General we can Use LinkedHashSet and LinkedHashMap for developing Cache Based Applications where Duplicates are Not Allowed. But Insertion Order Must be Preserved.

IdentityHashMap

- It is Exactly Same as HashMap Except the following Difference.
- In HashMap JVM will Use .equals() to Identify Duplicate Keys, which is Meant for ContentComparision.
- In IdentityHashMap JVM will Use == Operator to Identify Duplicate Keys, which is Meant for Reference Comparison.

What is the Difference between == Operator and .equals()?

== Operator

- Compares references (memory addresses) for objects.
- For primitive types (int, char, etc.), it compares actual values.
- For objects, it checks if both references point to the same object in memory

.equals() Method

- Defined in Object class; can be overridden by classes.
- Compares content (logical equality) if overridden.
- For String, Integer, and most wrapper classes, .equals() checks value equality.

Special Case

- For String literals, == may return true because of string pool
- For custom classes, override .equals() for meaningful comparison

```java
import java.util.HashMap;
import java.util.IdentityHashMap;
import java.util.Map;
@SuppressWarnings("removal")
public class IdentityHashMapDemo {

    Run | Debug
    public static void main(String[] args) {

        Map<Integer, String> h = new HashMap<>();

        IdentityHashMap<Integer, String> m = new IdentityHashMap<>();

        Integer I1 = new Integer(value: 10);
        Integer I2 = new Integer(value: 10);

        h.put(I1, value: "Pawan");
        h.put(I2, value: "Kalyan");

        m.put(I1, value: "Pawan");
        m.put(I2, value: "Kalyan");

        System.out.println("identity hashmap: "+m);
        System.out.println("hashmap: "+h);
    }
}
```

OUTPUT:

```
identity hashmap: {10=Kalyan, 10=Pawan}
hashmap: {10=Kalyan}
```

Here difference in output Because I1 and I2 are Not Duplicate as **I1 == I2** Returns false.

WeakHashMap

- In Case of HashMap, HashMap Dominates Garbage Collector. That is if Object doesn't have any Reference Still it is Not Eligible for Garbage Collector if it is associated with HashMap.
- But In Case of WeakHashMap if an Object doesn't contain any References, then it is Always Eligible for GC Even though it is associated with WeakHashMap. That is Garbage Collector Dominates WeakHashMap.

Use Cases

- Caches: Store data that should be discarded when the key object is no longer in use.
- Metadata: Attach extra info to objects without preventing their GC.

SortedMap

- It is the Child Interface of Map.
- If we want to Represent a Group of Key - Value Pairs According Some Sorting Order of Keys then we should go for SortedMap.

Methods

Object firstKey()

- Purpose: Returns the lowest key in the map according to the sorting order.

Object lastKey()

- Purpose: Returns the highest key in the map.

SortedMap headMap(Object key)

- Purpose: Returns a view of the portion of the map whose keys are strictly less than key.

SortedMap tailMap(Object key)

- Purpose: Returns a view of the portion of the map whose keys are greater than or equal to key.

SortedMap subMap(Object key1, Object key2)

- Purpose: Returns a view of the portion of the map whose keys range from key1 (inclusive) to key2 (exclusive).

Comparator comparator()

- Purpose: Returns the Comparator used to sort the keys, or null if natural ordering is used.

NavigableMap(I)

- A sorted map with additional navigation capabilities.

Methods:

floorKey(K key)

- Returns the greatest key ≤ given key.

lowerKey(K key)

- Returns the greatest key < given key.

ceilingKey(K key)

- Returns the least key ≥ given key.

higherKey(K key)

- Returns the least key > given key.

pollFirstEntry()

- Removes and returns the first (lowest) entry.

pollLastEntry()

- Removes and returns the last (highest) entry.

descendingMap()

- Returns a reverse-order view of the map.

TreeMap

- The Underlying Data Structure is Red -Black Tree.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Some Sorting Order of Keys.
- If we are depending on Default Natural Sorting Order, then the Keys should be Homogeneous and Comparable. Otherwise, we will get Runtime Exception Saying ClassCastException.
- If we are defining Our Own Sorting by Comparator, then Keys can be Heterogeneous and Non-Comparable. But there are No Restrictions on Values. They can be Heterogeneous and Non-Comparable.

null Acceptance

- For Empty TreeMap as the 1st Entry with null Key is Allowed. But After inserting that Entry if we are trying to Insert any Other Entry we will get RE: NullPointerException.
- For Non- Empty TreeMap if we are trying to Insert null Entry then we will get Runtime Exception Saying NullPointerException.
- There are No Restrictions on null Values.

Methods

TreeMap t = new TreeMap();

- **TreeMap<Integer, String> map = new TreeMap<>();**
- **Purpose**: Creates an empty TreeMap with natural ordering of keys.
- **Requirement**: Keys must implement Comparable.

TreeMap t = new TreeMap(Comparator c);

- **TreeMap<Integer, String> map = new TreeMap<>(Comparator.reverseOrder());**
- **Purpose**: Creates an empty TreeMap with a custom sorting order defined by the provided Comparator.

TreeMap t = new TreeMap(SortedMap m);

- **TreeMap<Integer, String> map = new TreeMap<>(sortedMap);**
- Purpose: Creates a new TreeMap containing all entries from the given SortedMap, preserving its order.

TreeMap t = new TreeMap(Map m);

- **Map<Integer, String> hashMap = new HashMap<>();**
- Purpose: Creates a new TreeMap containing all entries from the given Map, sorted by keys.

Hashtable(C)

- The Underlying Data Structure for Hashtable is Hashtable Only.
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.
- Heterogeneous Objects are Allowed for Both Keys and Values.
- Null Insertion is Not Possible for Both Key and Values. Otherwise, we will get Runtime Exception Saying NullPointerException.
- Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread Safe.

constructors

Hashtable h = new Hashtable();

- **Hashtable<Integer, String> table = new Hashtable<>();**
- **Purpose:** Creates an empty Hashtable with:
  - Default initial capacity: 11
  - Default load factor (fill ratio): 0.75
- Behavior: Automatically resizes when size exceeds capacity × loadFactor.

Hashtable h = new Hashtable(int initialCapacity);

- **Hashtable<Integer, String> table = new Hashtable<>(50);**
- Purpose: Creates an empty Hashtable with the specified initial capacity and default load factor.
- Use Case: When you know the approximate number of entries to avoid frequent resizing.

Hashtable h = new Hashtable(int initialCapacity, float fillRatio);

- **Hashtable<Integer, String> table = new Hashtable<>(100, 0.9f);**
- Purpose: Creates an empty Hashtable with:
  - Custom initial capacity
  - Custom load factor (fill ratio)
- Load Factor: Determines when to resize:

- Higher load factor → fewer resizes, more collisions
- Lower load factor → more space, faster lookups

Hashtable h = new Hashtable(Map m);

- **Hashtable<Integer, String> table = new Hashtable<>(map);**
- Purpose: Creates a new Hashtable and copies all entries from the given Map.

Properties

Properties(C)

The java.util.Properties class is a specialized subclass of Hashtable, designed to store key-value pairs where both the keys and the values are String objects. This makes it an ideal choice for managing configuration settings, such as those found in .properties files used by Java applications.

Key Features

- Type Restriction: Both keys and values must be Strings.
- Primary Purpose: Commonly used to read from and write to configuration files in Java applications.
- Thread Safety: Inherits synchronization from Hashtable, ensuring thread-safe operations.
- String-based Storage: Specifically designed to manage String-based key-value pairs.
- Configuration File Support: Provides convenient methods to load properties from input streams and store them to output streams, simplifying the process of working with configuration files.
- Default Values: Supports specifying default values when retrieving properties, allowing for flexible and robust configuration management.

Methods

Properties props = new Properties();

public String getProperty(String pname)

- props.getProperty(key);
- Purpose: Retrieves the value associated with the specified property name.
- Returns: The value as a String, or null if the property is not found.

public String setProperty(String pname, String pvalue)

- props.setProperty("password", "12345");
- Purpose: Adds or updates a property with the given name and value.
- Returns: The previous value associated with the key, or null if none.

public Enumeration propertyNames()

- Purpose: Returns an Enumeration of all property names (keys).

public void load(InputStream is)

- Purpose: Loads properties from a .properties file into the Properties object.

public void store(OutputStream os, String comment)

- Purpose: Stores properties from the Properties object into a file, with an optional comment.

Queue (I)

- Queue is a Child Interface of Collection.
- If we want to Represent a Group of Individual Objects Prior to processing, then we should go for Queue.
- From Version 1.5 onwards LinkedList also implements Queue Interface.
- Usually, Queue follows FIFO Order. But Based on Our Requirement we can Implement Our Own Priorities Also (PriorityQueue)
- LinkedList based Implementation of Queue always follows FIFO Order.
- Eg: Before sending a Mail, we have to Store all Mail IDs in Some Data Structure and for the 1st Inserted Mail ID Mail should be Sent 1st.For this Requirement Queue is the Best Choice.

Methods

Queue<Integer> q = new LinkedList<>();

boolean offer(Object o)

- q.offer(10);
- Purpose: Adds an element to the queue.
- Returns: true if the element was added successfully, false otherwise.
- Behavior: Does not throw an exception if the queue is full (for bounded queues).

Object peek()

- q.peek()
- Purpose: Retrieves (but does not remove) the head element of the queue.
- Returns: Head element, or null if the queue is empty.

Object element()

- q.element()
- Purpose: Retrieves (but does not remove) the head element of the queue.
- Difference from peek(): Throws NoSuchElementException if the queue is empty.

Object poll()

- q.poll()
- Purpose: Removes and returns the head element of the queue.
- Returns: Head element, or null if the queue is empty.

Object remove()

- q2.remove()
- Purpose: Removes and returns the head element of the queue.
- Difference from poll(): Throws NoSuchElementException if the queue is empty.

PriorityQueue

- This is a Data Structure which can be used to Represent a Group of Individual Objects Prior to processing according to Some Priority.
- The Priority Order can be Either Default Natural Sorting Order OR Customized Sorting Order specified by Comparator Object.
- If we are Depending on Natural Sorting Order, then the Objects should be Homogeneous and Comparable otherwise we will get ClassCastException.

- If we are defining Our Own Sorting by Comparator, then the Objects Need Not be Homogeneous and Comparable.
- Duplicate objects are Not Allowed.
- Insertion Order is Not Preserved and it is Based on Some Priority.
- null Insertion is Not Possible Even as 1st Element Also.

constructors

PriorityQueue q = new PriorityQueue();

- Purpose: Creates an empty PriorityQueue with:
- Default initial capacity: 11
- Default ordering: Natural ordering of elements (requires elements to implement Comparable).

PriorityQueue q = new PriorityQueue(int initialCapacity);

- Purpose: Creates an empty PriorityQueue with the specified initial capacity and natural ordering.
- Use Case: When you know the approximate number of elements to avoid resizing.

PriorityQueue q = new PriorityQueue(int initialCapacity, Comparator c);

- Purpose: Creates an empty PriorityQueue with:
  - Custom initial capacity
  - Custom ordering defined by the provided Comparator

PriorityQueue q = new PriorityQueue(SortedSet s);

- Purpose: Creates a PriorityQueue containing all elements from the given SortedSet, preserving its order.

PriorityQueue q = new PriorityQueue(Collection c);

- Purpose: Creates a PriorityQueue containing all elements from the given collection.

BlockingQueue(I)

- It is the Child Interface of Queue. Present in java.util.Concurrent Package.
- It is a Thread Safe Collection.
- It is a specially designed Collection Not Only to Store Elements but also Supports Flow Control by Blocking Mechanism.
- If Queue is Empty take() (Retrieval Operation) will be Blocked until Queue will be Updated with Items.
- put() will be blocked if Queue is Full until Space Availability.
- This Property Makes BlockingQueue Best Choice for Producer Consumer Problem. When
- One Thread producing Items to the Queue and the Other Thread consuming Items from the Queue.

TransferQueue(I)

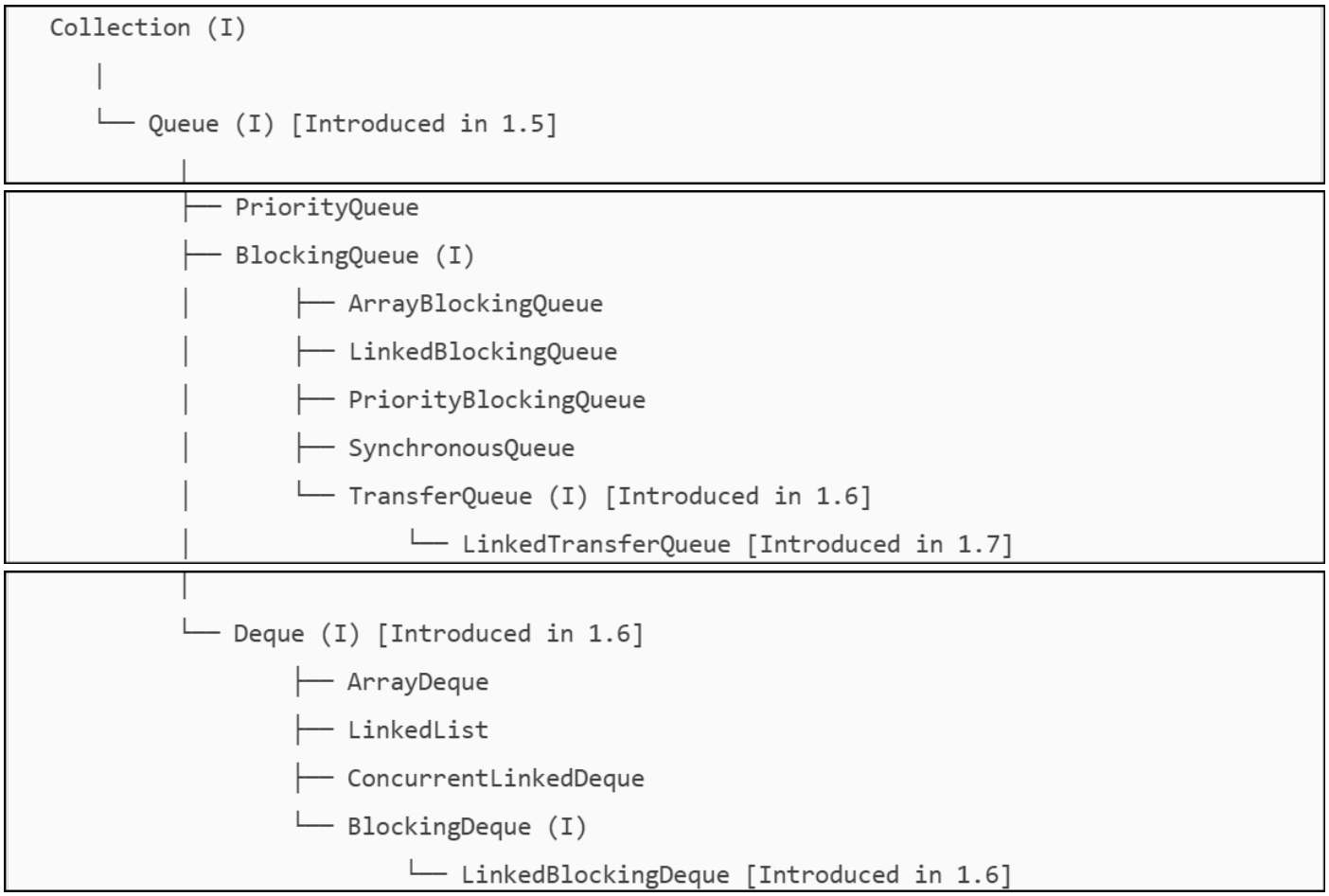
- In BlockingQueue we can Only Put Elements into the Queue and if Queue is Full then Our put() will be blocked until Space is Available.
- But in TransferQueue we can also Block until Other Thread receiving Our Element. Hence this is the Behaviour of transfer().
- In BlockingQueue we are Not required to wait until Other Threads Receive Our Element
- but in TransferQueue we have to wait until Some Other Thread Receive Our Element.
- TrasferQueue is the Best Choice for Message Passing Application where Guarantee for the Delivery

Deque (I)

- It Represents a Queue where we can Insert and Remove Elements from Deque, Both Ends of Queue i.e. Deque Means Double Ended Queue.
- It is Also pronounced as Deck Like Deck of Cards.

BlockingDeque (I) 1.6 V

- It is the Child Interface of **BlockingQueue** and **Deque**.
- It is a Simple Deque with Blocking Operations but wait for the Deque to become Non Empty for Retrieval Operation and wait for Space to Store Element.



Concurrent Collections (1.5)

Need for Concurrent Collections

- Tradition Collection Object (Like ArrayList, HashMap Etc) can be accessed by Multiple Threads simultaneously and there May be a Chance of Data Inconsistency Problems and Hence these are Not Thread Safe.
- Already existing Thread Safe Collections (Vector, Hashtable, synchronizedList(), synchronizedSet(), synchronizedMap()) Performance wise Not Up to the Mark.
- Because for Every Operation Even for Read Operation Also Total Collection will be loaded by Only One Thread at a Time and it Increases waiting Time of Threads.
- Another Big Problem with Traditional Collections is while One Thread iterating Collection, the Other Threads are Not allowed to Modify Collection Object simultaneously if we are trying to Modify then we will get ConcurrentModificationException.

❖ Hence these Traditional Collection Objects are Not Suitable for Scalable Multi-Threaded Applications.

```java
import java.util.ArrayList;
import java.util.Iterator;

class Test {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("A");
        al.add("B");
        al.add("C");

        Iterator itr = al.iterator();
        while (itr.hasNext()) {
            String s = (String) itr.next();
            System.out.println(s);
            // al.add("D"); // Uncommenting this causes ConcurrentModificationException
        }
    }
}
```

To Overcome these Problems SUN People introduced Concurrent Collections in 1.5 Version.

1. Concurrent Collections are Always Thread Safe.
2. When compared with Traditional Thread Safe Collections Performance is More because of different Locking Mechanism.
3. While One Thread interacting Collection the Other Threads are allowed to Modify Collection in Safe Manner.

Hence Concurrent Collections Never threw ConcurrentModificationException.

The Important Concurrent Classes are

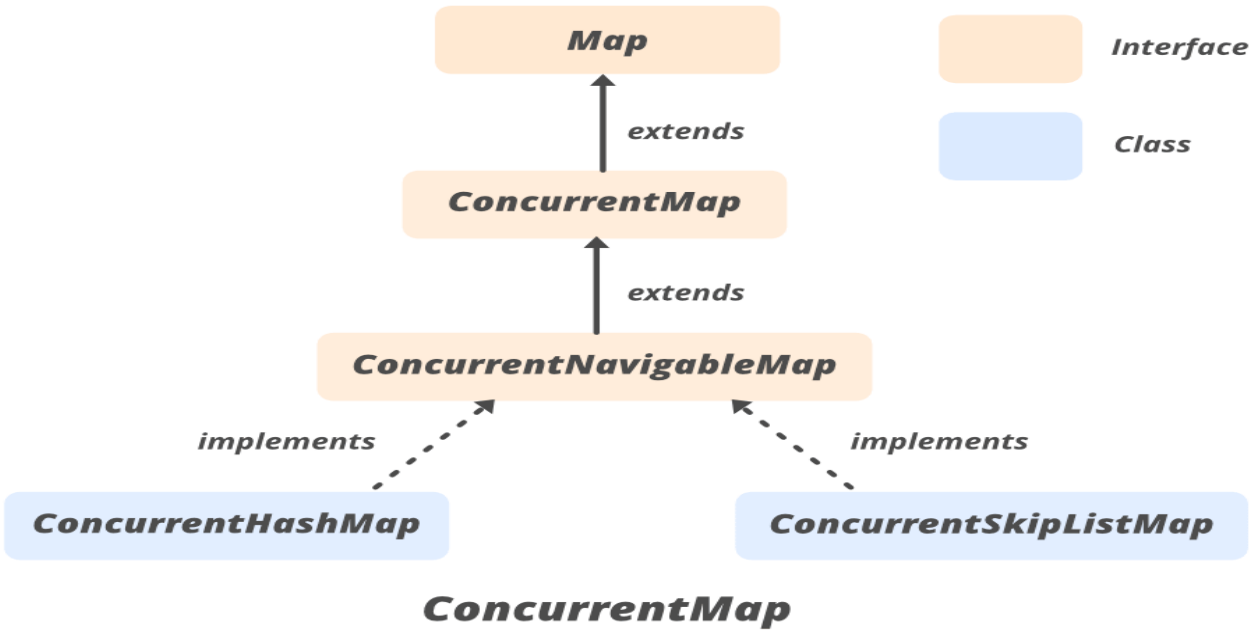
❖ ConcurrentHashMap

❖ CopyOnWriteArrayList

❖ CopyOnWriteArraySet

ConcurrentMap (I)

❖ ConcurrentMap is part of the **java.util.concurrent** package and is an extension of the **Map** interface. It provides thread-safe operations for maps without requiring explicit synchronization.



- ConcurrentMap is part of the java.util.concurrent package and is an extension of the Map interface. It provides thread-safe operations for maps without requiring explicit synchronization.
- This is especially useful in concurrent programming where multiple threads access and modify the map simultaneously.

Methods:

putIfAbsent(K key, V value)

- **Purpose**: Inserts the specified key-value pair only if the key is not already associated with a value.
- **Behaviour**:
  - If the key is absent, it adds the new mapping and returns null.
  - If the key is present, it does nothing and returns the existing value.
- **Use Case**: Avoid overwriting existing values in concurrent environments.

remove(Object key, Object value)

- **Purpose**: Removes the entry for the specified key only if it is currently mapped to the given value.
- **Behaviour**:
  - Returns true if removal was successful.
  - Returns false if the key does not exist or is mapped to a different value.
- **Use Case**: Safe removal without race conditions.

replace(K key, V oldValue, V newValue)

- **Purpose**: Replaces the value for the specified key only if currently mapped to oldValue.
- **Behaviour:**
  - Returns true if replacement was successful.
  - Returns false if the key does not exist or is mapped to a different value.
- **Use Case:** Atomic conditional update.

ConcurrentHashMap(C)

- Underlying Data Structure is Hashtable.
- ConcurrentHashMap allows Concurrent Read and Thread Safe Update Operations.
- To Perform Read Operation Thread won't require any Lock. But to Perform Update Operation Thread requires Lock but it is the Lock of Only a Particular Part of Map (Bucket Level Lock).
- Instead of Whole Map Concurrent Update achieved by Internally dividing Map into Smaller Portion which is defined by Concurrency Level. The Default Concurrency Level is 16.
- That is ConcurrentHashMap Allows simultaneous Read Operation and simultaneously 16 Write (Update) Operations.
- null is Not Allowed for Both Keys and Values.
- While One Thread iterating the Other Thread can Perform Update Operation and ConcurrentHashMap Never throw ConcurrentModificationException.

Constructors:

ConcurrentHashMap m = new ConcurrentHashMap();

- Creates an empty ConcurrentHashMap with Default initial capacity 16, Default load factor 0.75 and Default concurrency level: 16

ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity);

- Creates a ConcurrentHashMap with the given initial capacity.

ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio);

- Creates a ConcurrentHashMap with Initial capacity and Load factor (fillRatio): Determines when to resize.

ConcurrentHashMap m =

new ConcurrentHashMap(int initialCapacity, float fillRatio, int concurrencyLevel);

- Creates a ConcurrentHashMap with Initial capacity, Load factor and Concurrency level: Number of threads that can modify the map concurrently without contention.

ConcurrentHashMap m = new ConcurrentHashMap(Map m);

- Creates a new ConcurrentHashMap and initializes it with the mappings from the specified map.

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.*;

class MyThread extends Thread {
    // If we use HashMap instead of ConcurrentHashMap, it will throw ConcurrentModificationException
    // static HashMap<Integer, String> m = new HashMap<>();
    static ConcurrentHashMap<Integer, String> m = new ConcurrentHashMap<>();

    public void run() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Child Thread updating Map");
        m.put(103, "C");
    }
```

```
    public static void main(String[] args) throws InterruptedException {
        m.put(101, "A");
        m.put(102, "B");

        MyThread t = new MyThread();
        t.start();

        Set<Integer> s = m.keySet();
        Iterator<Integer> itr = s.iterator();

        while (itr.hasNext()) {
            Integer l1 = itr.next();
            System.out.println("Main Thread iterating and Current Entry is: " + l1 + " ..... " + m.get(l1));
            Thread.sleep(3000);
        }

        System.out.println(m);
    }
}
```

Output:

```
PS C:\Users\61093419\OneDrive - LTIMindtree\Documents\Codes> javac MyThread.java
PS C:\Users\61093419\OneDrive - LTIMindtree\Documents\Codes> java MyThread
Main Thread iterating and Current Entry is: 101 ..... A
Child Thread updating Map
Main Thread iterating and Current Entry is: 102 ..... B
Main Thread iterating and Current Entry is: 103 ..... C
{101=A, 102=B, 103=C}
```

Reason

- In the Case of **ConcurrentHashMap** iterator creates a Read Only Copy of Map Object and iterates over that Copy if any Changes to the Map after getting iterator it won't be affected/ reflected.
- In the Above Program if we Replace ConcurrentHashMap with HashMap then we will get **ConcurrentModificationException**.

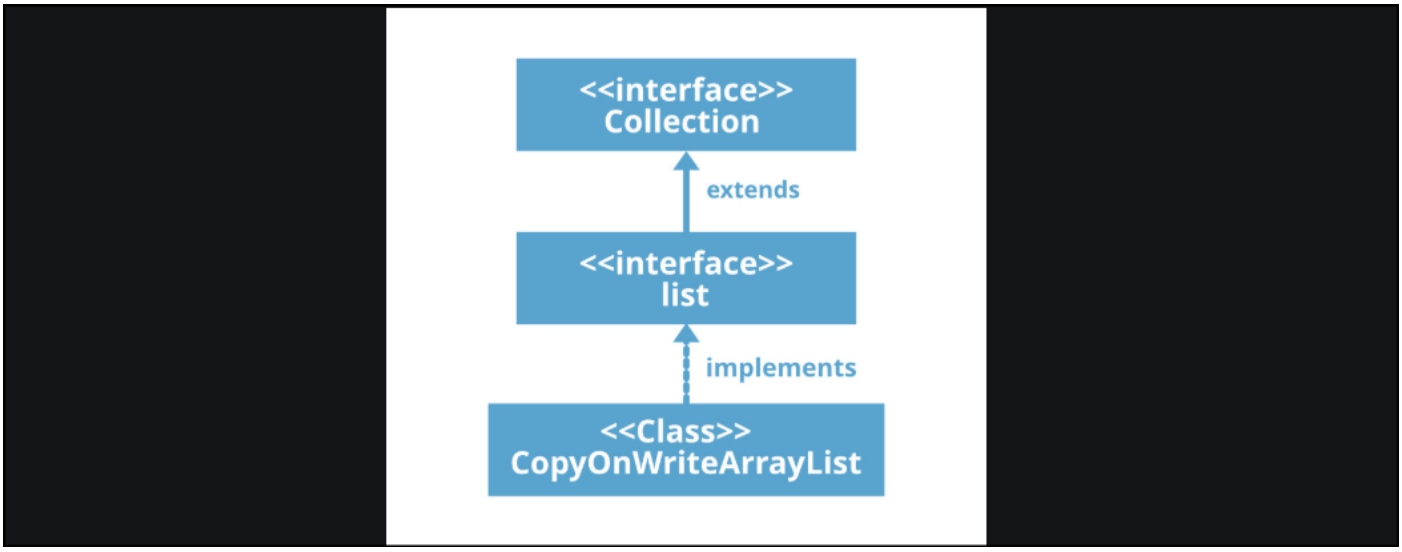
Difference between ConcurrentHashMap, synchronizedMap(), and Hashtable

Feature	**ConcurrentHashMap**	**synchronizedMap()**	**Hashtable**
Thread Safety Mechanism	Thread safety using **bucket-level locking** (locks only part of the map)	Thread safety by **locking the entire map object**	Thread safety by **locking the entire map object**
Concurrency	Multiple threads can operate safely at the same time	Only one thread can operate at a time	Only one thread can operate at a time
Read/Write Operations	Read operations without lock; write operations use bucket-level lock	Both read and write operations require full map lock	Both read and write operations require full map lock
Concurrent Modification During Iteration	Allowed safely; does **not throw ConcurrentModificationException**	Not allowed; throws **ConcurrentModificationException**	Not allowed; throws **ConcurrentModificationException**
Iterator Behaviour	**Fail-safe** (does not throw exception; weakly consistent)	**Fail-fast** (throws exception if modified during iteration)	**Fail-fast** (throws exception if modified during iteration)
Null Handling	Does **not allow null keys or values**	Allows **null keys and values**	Does **not allow null keys or values**
Version Introduced	Java **1.5**	Java **1.2**	Java **1.0**

Difference between HashMap and ConcurrentHashMap

Feature	HashMap	ConcurrentHashMap
Thread Safety	Not thread-safe	Thread-safe
Performance	Relatively high (no thread synchronization)	Relatively lower (due to internal synchronization)
Concurrent Modification	Throws ConcurrentModificationException if modified during iteration	Allows safe modification during iteration without exceptions
Iterator Behaviour	Fail-fast (throws exception on modification)	Fail-safe (does not throw exception; weakly consistent view)
Null Handling	Allows one null key and multiple null values	Does not allow null keys or values (throws NullPointerException)
Version Introduced	Java 1.2	Java 1.5

CopyOnWriteArrayList (C)



- It is a Thread Safe Version of ArrayList as the Name indicates CopyOnWriteArrayList Creates a Cloned Copy of Underlying ArrayList for Every Update Operation at Certain Point Both will Synchronized Automatically Which is taken Care by JVM Internally.
- As Update Operation will be performed on cloned Copy there is No Effect for the Threads which performs Read Operation.
- It is Costly to Use because for every Update Operation a cloned Copy will be Created. Hence CopyOnWriteArrayList is the Best Choice if Several Read Operations and a smaller Number of Write Operations are required to Perform.
- Insertion Order is Preserved, Duplicate Objects are allowed, Heterogeneous Objects are allowed, null Insertion is Possible.
- It implements Serializable, Cloneable and RandomAccess Interfaces.
- While One Thread iterating CopyOnWriteArrayList, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException. That is iterator is Fail Safe.
- Iterator of ArrayList can Perform Remove Operation but Iterator of CopyOnWriteArrayList can't Perform Remove Operation. Otherwise we will get RuntimeException Saying UnsupportedOperationException.

Constructors

CopyOnWriteArrayList l = new CopyOnWriteArrayList();

- Creates an empty list with the default initial capacity and useful when you want to start with an empty thread-safe list.

CopyOnWriteArrayList l = new CopyOnWriteArrayList(Collection c);

- Creates a list containing all elements of the specified collection then the elements are copied into a new internal array.

CopyOnWriteArrayList l = new CopyOnWriteArrayList(Object[] a);

- Creates a list containing all elements from the given array.

Methods

public boolean addIfAbsent(E e)

- Adds the element only if it is not already present in the list (checked via equals).
- This is atomic with respect to other concurrent operations—safe in multithreaded contexts without external synchronization.
- Return value:
  - true → the element was not present and has been added.
  - false → the element was already present; list remains unchanged.

public int addAllAbsent(Collection<? extends E> c)

- Adds all elements from c that are not already present in the list.
- Each element is checked against current contents; duplicates in c are ignored, and elements already in the list are not added.
- Operation is atomic with respect to other threads: it validates absence and performs a single write (array copy) for all new elements.
- Return value:
  - The number of elements added (i.e., absent ones).
  - Returns 0 if nothing new was added.

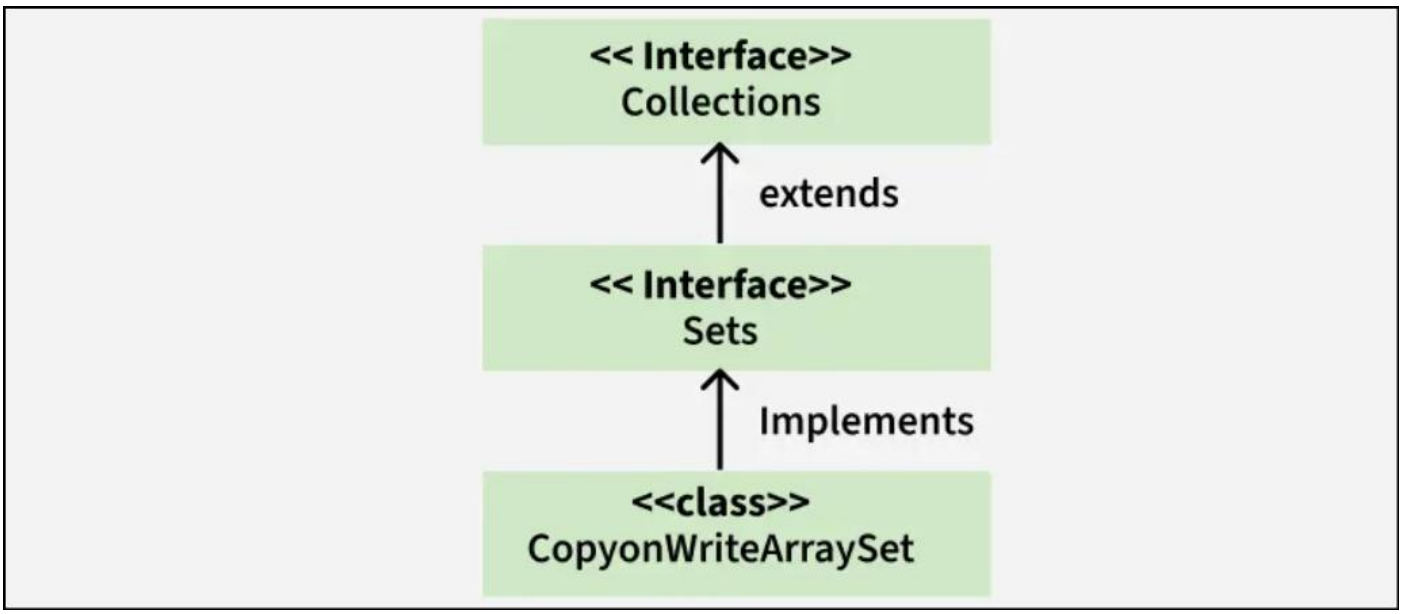
Differences between ArrayList and CopyOnWriteArrayList

Feature	ArrayList	CopyOnWriteArrayList
Thread Safety	Not Thread Safe	Not Thread Safe because Every Update Operation will be performed on Separate cloned Coy
Modification during Iteration	Other Threads are Not allowed to Modify List Otherwise we will get ConcurrentModificationException	Other Threads are allowed to Modify List in Safe Manner and we won't get ConcurrentModificationException
Iterator Type	Fail-Fast	Fail-Safe
Iterator Remove Operation	Iterator of ArrayList can Perform Remove Operation	Iterator of CopyOnWriteArrayList can't Perform Remove Operation Otherwise we will get RuntimeException: UnsupportedOperationException
Introduced in Version	1.2 Version	1.5 Version

Differences between CopyOnWriteArrayList, synchronizedList() and vector()

Feature	CopyOnWriteArrayList	synchronizedList()	vector()
Thread Safety	We will get Thread Safety because Every Update Operation will be performed on Separate cloned Copy.	We will get Thread Safety because at a Time List can be accessed by Only One Thread at a Time.	We will get Thread Safety because at a Time Only One Thread is allowed to Access Vector Object.
Concurrency	At a Time Multiple Threads are allowed to Access/Operate on CopyOnWriteArrayList.	At a Time Only One Thread is allowed to Perform any Operation on List Object.	At a Time Only One Thread is allowed to Operate on Vector Object.
Modification During Iteration	While One Thread iterating List Object, the Other Threads are allowed to Modify Map and we won't get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify List. Otherwise we will get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify Vector. Otherwise we will get ConcurrentModificationException.
Iterator Behaviour	Iterator is Fail-Safe and won't raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.	Iterator is Fail-Fast and it will raise ConcurrentModificationException.
Remove Operation in Iterator	Iterator can't Perform Remove Operation Otherwise we will get UnsupportedOperationException.	Iterator can Perform Remove Operation.	Iterator can Perform Remove Operation.
Version Introduced	Introduced in 1.5 Version.	Introduced in 1.2 Version.	Introduced in 1.0 Version.

CopyOnWriteArraySet(C)



- It is a Thread Safe Version of Set which is Internally Implement by CopyOnWriteArrayList.
- Insertion Order is Preserved and duplicate Objects are Not allowed.
- Multiple Threads can be Able to Perform Read Operation simultaneously but for Every Update Operation a Separate cloned Copy will be Created.
- As for Every Update Operation a Separate cloned Copy will be Created which is Costly Hence if Multiple Update Operation are required then it is Not recommended to Use CopyOnWriteArraySet.
- While One Thread iterating Set the Other Threads are allowed to Modify Set and we won't get **ConcurrentModificationException**.
- Iterator of **CopyOnWriteArraySet** can Perform Only Read Operation and won't Perform Remove Operation. Otherwise we will get RuntimeException: **UnsupportedOperatonException**.

Constructors:

CopyOnWriteArraySet s = new CopyOnWriteArraySet();

- Creates an empty CopyOnWriteArraySet which is Internally backed by a CopyOnWriteArrayList.

CopyOnWriteArraySet s = new CopyOnWriteArraySet(Collection c);

- Creates a set containing all elements from the specified collection. Duplicate elements in the collection are ignored.

Methods

Whatever Methods Present in Collection and Set Interfaces are the Only Methods Applicable for CopyOnWriteArraySet and there are No Special Methods.

Differences between CopyOnWriteArraySet and synchronizedSet()

Feature	CopyOnWriteArraySet()	synchronizedSet()
Thread Safety	It is Thread Safe because Every Update Operation will be performed on Separate Cloned Copy.	It is Thread Safe because at a Time Only One Thread can Perform Operation.
Modification During Iteration	While One Thread iterating Set, the Other Threads are allowed to Modify and we won't get ConcurrentModificationException.	While One Thread iterating, the Other Threads are Not allowed to Modify Seta Otherwise we will get ConcurrentModificationException.
Iterator Behaviour	Iterator is Fail Safe.	Iterator is Fail Fast.
Iterator Operations	Iterator can Perform Only Read Operation and can't Perform Remove Operation Otherwise we will get RuntimeException Saying UnsupportedOperationException.	Iterator can Perform Both Read and Remove Operations.
Version Introduced	Introduced in 1.5 Version.	Introduced in 1.7 Version.

Fail Fast Vs Fail Safe Iterators

Fail Fast Iterator:

- While One Thread iterating Collection if Other Thread trying to Perform any Structural Modification to the underlying Collection then immediately Iterator Fails by raising **ConcurrentModificationExcepion**. Such Type of Iterators are Called Fail Fast Iterators.
- Internally Fail Fast Iterator will Use Some Flag named with MOD to Check underlying Collection is Modified OR Not while iterating.

Fail Safe Iterator:

- While One Thread iterating if the Other Threads are allowed to Perform any Structural Changes to the underlying Collection, Such Type of Iterators are Called Fail Safe Iterators.
- Fail Safe Iterators won't raise ConcurrentModificationException because Every Update Operation will be performed on Separate cloned Copy.

Fail-Fast vs Fail-Safe Iterators

Feature	Fail-Fast Iterator	Fail-Safe Iterator
Definition	Immediately fails by throwing ConcurrentModificationException if the collection is modified while iterating.	Works on a **clone or snapshot** of the collection, so modifications do not affect the iterator.
Examples	ArrayList, HashMap, Vector, Collections.synchronizedList()	CopyOnWriteArrayList, CopyOnWriteArraySet, ConcurrentHashMap
Behaviour on Modification	Throws ConcurrentModificationException	Does **not throw exception**; changes are not reflected in the iterator.
Underlying Mechanism	Directly references the original collection.	Iterates over a **separate copy** of the collection.
Performance	Faster (no extra copy), but unsafe in concurrent modifications.	Slower for updates (due to copying), but safe for concurrent access.
Iterator Type	Fail-Fast	Fail-Safe
Remove Operation	Supported (via iterator's remove() method).	Not supported; calling remove() throws UnsupportedOperationException.

Enum with Collections

EnumSet

- It is a specially designed Set implemented Collection Applicable Only for Enum which is Introduced in 1.5 Version.
- EnumSet is Internally implemented as Bit Vectors which Improves Performance Internally.
- The Performance of EnumSet is Very High if we want to Store Enum Constants than Traditional Collections (Like HashSet, LinkedHashSetEtc).
- All Elements of the EnumSet should be from Same Enum Type Only if we are trying to Add Elements from different enums then we will get Compile Time Error (i.e. EnumSet is Type Safe Collection).
- Iterator Returned by EnumSet Traverse, Iterate Elements in their Natural Order i.e. the Order in which the Enum Constants are declared i.e. the Order Returned by ordinal().
- Enum Iterator Never throw Concurrent Modification Exception.
- Inside EnumSet we can't Add null Otherwise we will get NullPointerException.
- EnumSet is an Abstract Class and Hence we can't Create Object directly by using new Key Word.
- EnumSet defined Several Factory Methods to Create EnumSet Object.
- EnumSet defines 2 Child Classes.
    - RegularEnumSet
    - JumboEnumSet

- The Factory Methods will Return this Class Objects Internally Based on Size if the Size is < 64 then RegularEnumSet will be choose Otherwise if Size > 64 then JumboEnumSet will be choose.

EnumMap

- It is a specially designed Map to Use Enum Type Objects as Keys which is Introduced in 1.5 Version.
- It implements Serializable and Cloneable Interfaces.
- EnumMap is Internally implemented by using Bit Vectors (Arrays), which Improves Performance when compared with Traditional Map Object Like HashMap Etc.
- All Keys to the EnumMap should be from a Single Enum if we are trying to Use from different Enum then we will get Compile Time Error. Hence EnumMap is Type Safe.
- Iterator Never throw **ConcurrentModificationException**.
- Iterators of EnumMap iterate Elements according to Ordinal Value of Enum Keys i.e. in which Order Enum Constants are declared in the Same Order Only Iterator will be iterated.
- null Key is Not allowed Otherwise we will get NullPointerException.

Constructors

EnumMap m = new EnumMap(Class KeyType)

- Purpose: Creates an empty EnumMap with the specified enum key type.
- Key Requirement: The Class KeyType must be an enum class.

EnumMap m = new EnumMap(EnumMap m1)

- Purpose: Creates a new EnumMap with the same key type as the given EnumMap and copies all mappings.

EnumMap m = new EnumMap(Map m1)

- Purpose: Creates an EnumMap equivalent to the given Map.
- Condition: The keys in m1 must be of the same enum type.

Methods

- EnumMap doesn't contain any New Methods. We have to Use General Map Methods Only.